Open in app ↗

Search

Distributed Cache

Suresh Podeti

8 min read · Sep 23, 2023

Listen Share More

Introduction

Cache

A cache is a **non-persistent storage** area used to keep **repeatedly read** and written data, which provides the end user with **lower latency**. Therefore, a cache must serve data from a storage component that is fast, has enough storage, and is affordable in terms of dollar cost as we scale the caching service. The following illustration highlights the suitability of **RAM** as the raw building block for caching:

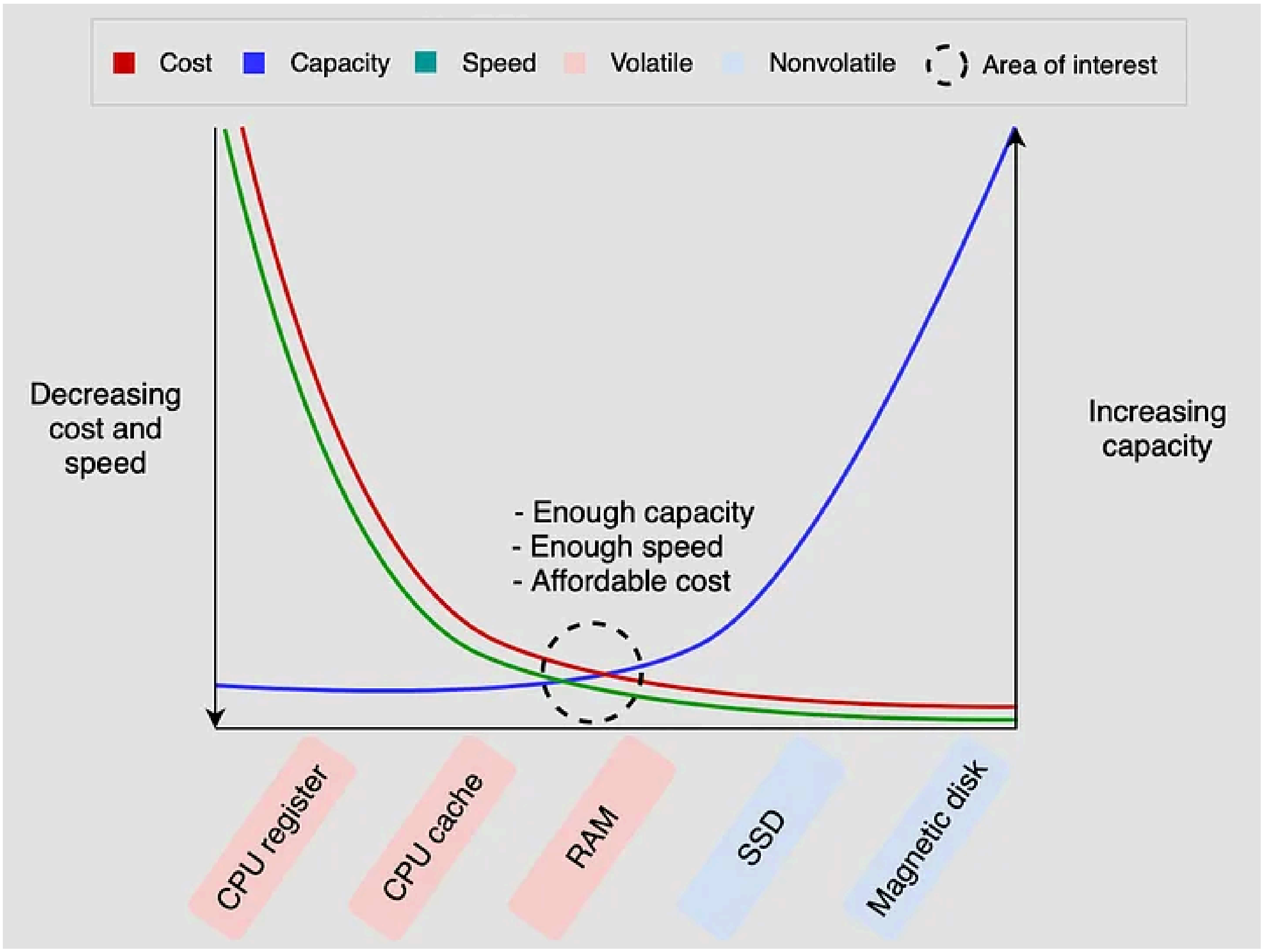

Fig 1.0: An approximation that depicts how RAM is the optimal choice for serving cached data

Distributed Cache

A **distributed cache** is a caching system where **multiple cache servers coordinate to store frequently accessed data**. Distributed caches are needed in environments where a **single cache server isn't enough** to store all the data. At the same time, it's **scalable** and guarantees a higher degree of **availability**.

Background

Writing policies

Data is written to cache and databases. The **order** in which data writing happens has **performance implications.**

- **Write-around cache:** This strategy involves writing data to the database only. Later, when a read is triggered for the data, it's written to cache after a cache miss. The database will have updated data, but such a strategy isn't favourable for reading recently updated data.
- **Write-back cache:** In the write-back cache mechanism, the data is first written to the cache and asynchronously written to the database. Although the cache

has updated data, inconsistency is inevitable in scenarios where a client reads stale data from the database. However, systems using this strategy will have small writing latency.

- **Write-through cache**: The write-through mechanism writes on the cache as well as on the database. Writing on both storages can happen concurrently or one after the other. This increases the write latency but ensures strong consistency between the database and the cache.

Eviction policies

One of the main reasons caches perform fast is that they're small. Small caches mean **limited storage capacity**. Therefore, we need an eviction mechanism to **remove less frequently accessed** data from the cache.

The most well-known strategies include the following:

- **Least recently used (LRU)**
- **Most recently used (MRU)**
- **Least frequently used (LFU)**
- **Most frequently used (MFU)**
- **First-in-First-Out (FIFO)**

Cache invalidation

Apart from the eviction of less frequently accessed data, some data residing in the cache may become **stale or outdated over time**. Such cache entries are invalid and must be marked for deletion.

The situation demands a question: How do we identify stale entries?

Resolution of the problem requires storing metadata corresponding to each cache entry. Specifically, maintaining a time-to-live (TTL) value to deal with outdated cache items.

We can use two different approaches to deal with outdated items using **TTL**:

- **Active expiration:** This method actively checks the TTL of cache entries through a daemon process or thread.

- **Passive expiration**: This method checks the TTL of a cache entry at the time of access.

Each expired item is removed from the cache upon discovery.

Storage mechanism

When we use multiple cache servers, the following design questions need to be answered:

- Which data should we store in which cache servers?
- What data structure should we use to store the data?

Hash function

It's possible to use hashing in two different scenarios:

- Identify the cache server in a distributed cache to store and retrieve data.
- Locate cache entries inside each cache server.

For the first scenario, we can use **consistent hashing** or its flavors, because simple hashing won't be ideal in case of crashes or scaling.

In the second scenario, we can use typical **hash functions** to locate a cache entry to read or write inside a cache server.

Linked list

However, a hash function alone can only locate a cache entry. It doesn't say anything about managing data within the cache server. That is, it doesn't say anything about how to implement a strategy to evict less frequently accessed data from the cache server.

We'll use a **doubly linked list**. The main reason is its widespread usage and simplicity. Furthermore, **adding and removing data** from the doubly linked list in our case will be a **constant time operation**. This is because we either evict a specific entry from the tail of the linked list or relocate an entry to the head of the doubly linked list. Therefore, no iterations are required.

Bloom filters are an interesting choice for quickly finding if a cache entry doesn't exist in the cache servers, but it comes with probabilistic models.

Requirements

- **High performance:** The primary reason for the cache is to enable fast retrieval of data. Therefore, both the `insert` and `retrieve` operations must be fast.
- **Scalability:** The cache system should scale horizontally with no bottlenecks on an increasing number of requests.
- **High availability:** The unavailability of the cache will put an extra burden on the database servers, which can also go down at peak load intervals. We also require our system to survive occasional failures of components and network, as well as power outages.
- **Consistency:** Data stored on the cache servers should be consistent. For example, different cache clients retrieving the same data from different cache servers (primary or secondary) should be up to date.

Design

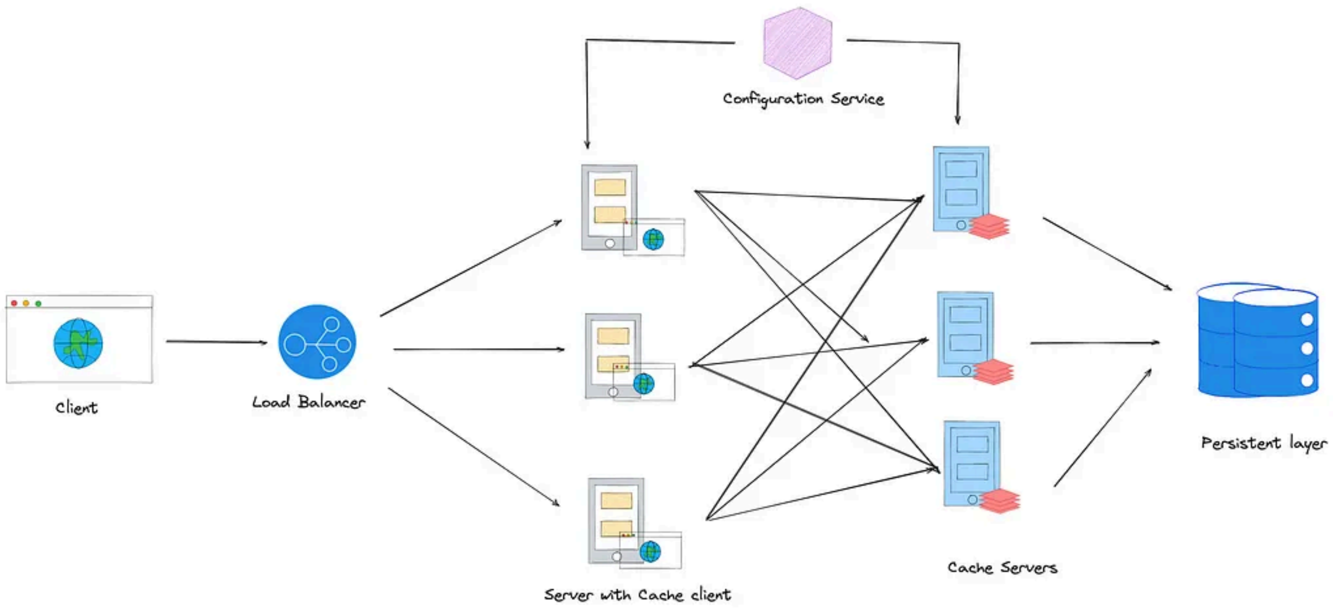

Fig 2.0: High-level design of distributed cache

The main components in this high-level design are the following:

- The client's requests reach the service hosts through the load balancers where the cache clients reside.
- Each cache client uses consistent hashing to identify the cache server. Next, the cache client forwards the request to the cache server maintaining a specific

shard.

- **Cache servers**: These servers maintain the cache of the data. Each cache server is accessible by all the cache clients. Each server is connected to the database to store or retrieve data. Cache clients use **TCP or UDP protocol** to perform data transfer to or from the cache servers. However, if any cache server is down, requests to those servers are resolved as a missed cachpe by the cache clients.

- **Configuration service:** continuously monitors the health of the cache servers. In addition to that, the **cache clients will get notified** when a new cache server is added to the cluster. When we use this strategy, no human intervention or monitoring will be required in case of failures or the addition of new nodes. Finally, the cache clients obtain the list of cache servers from the configuration service.

- Each cache server has primary and replica servers. Internally, every server uses the same mechanisms to store and evict cache entries.

Internals of cache server

Each cache client should use three mechanisms to **store** and **evict** entries from the cache servers:

- **Hash map**: The cache server uses a hash map to store or locate different entries inside the RAM of cache servers.

- **Doubly linked list**: If we have to evict data from the cache, we require a linked list so that we can order entries according to their frequency of access.

- **Eviction policy**: The eviction policy depends on the application requirements. Here, we assume the least recently used (LRU) eviction policy.

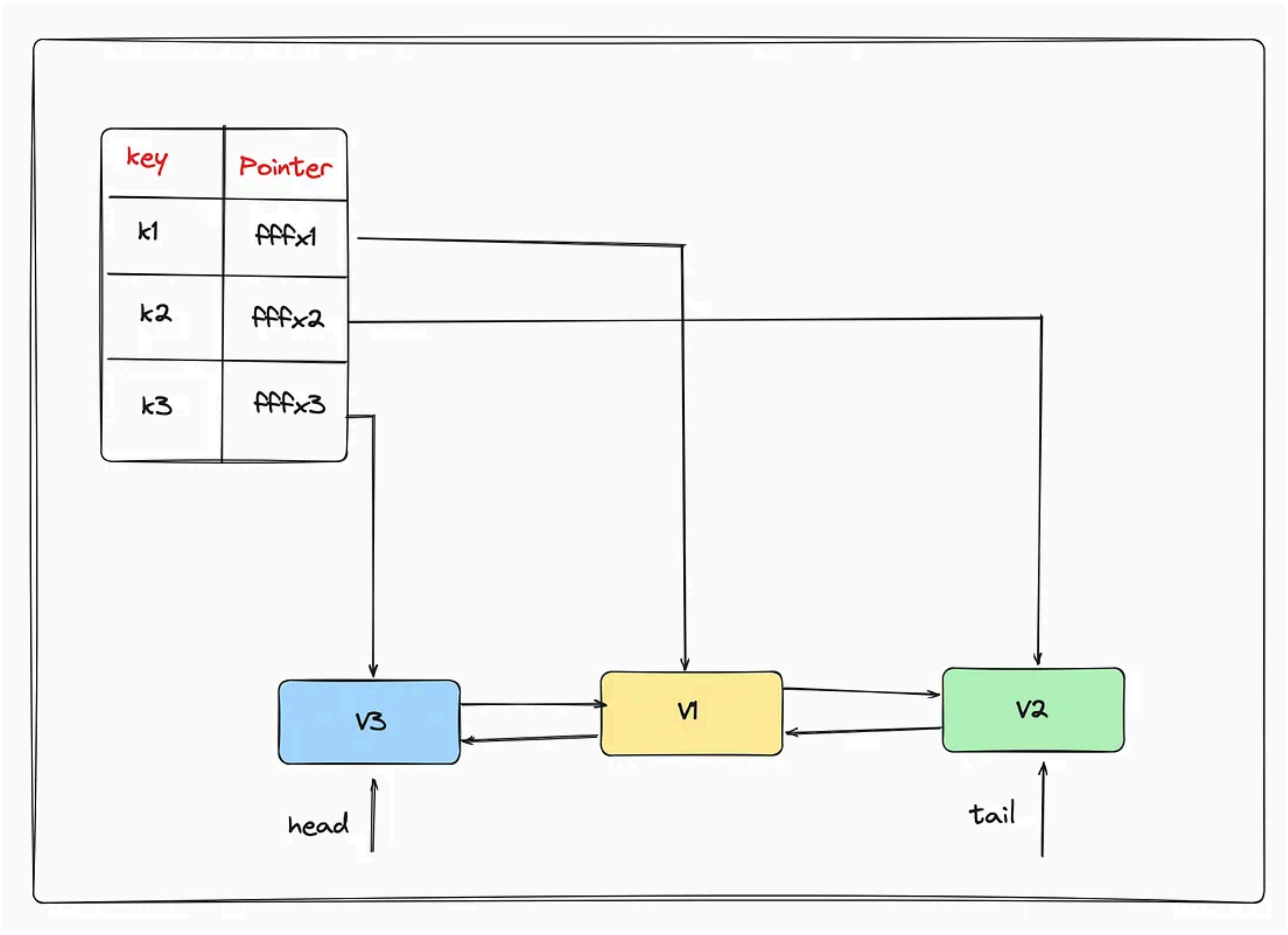

Fig 3.0: Internals of cache server

Evaluation

performance

Here are some design choices we made that will contribute to overall good performance:

- We used consistent hashing. Finding a key under this algorithm requires a time complexity of O(log(N)), where N represents the number of cache shards.
- Inside a cache server, keys are located using hash tables that require constant time on average.
- The LRU eviction approach uses a constant time to access and update cache entries in a doubly linked list.
- The communication between cache clients and servers is done through TCP and UDP protocols, which is also very fast.

- Since we added more replicas, these can reduce the performance penalties that we have to face if there's a high request load on a single machine.
- An important feature of the design is adding, retrieving, and serving data from the RAM. Therefore, the latency to perform these operations is quite low.

Scalability

We can create shards based on requirements and changing server loads. While we add new cache servers to the cluster, we also have to do a limited number of rehash computations, thanks to consistent hashing.

Adding replicas reduces the load on hot shards. Another way to handle the hotkeys problem is to do further sharding within the range of those keys. Although the scenario where a single key will become hot is rare, it's possible for the cache client to devise solutions to avoid the single hotkey contention issue.

Availability

We have improved the availability through redundant cache servers. Redundancy adds a layer of reliability and fault tolerance to our design. We also used the leader-follower algorithm to conveniently manage a cluster shard. However, we haven't achieved high availability because we have two shard replicas, and at the moment, we assume that the replicas are within a data center.

It's possible to achieve higher availability by splitting the leader and follower servers among different data centers. But such high availability comes at a price of consistency. We assumed synchronous writes within the same data center. But synchronous writing for strong consistency in different data centers has a serious performance implication that isn't welcomed in caching systems. We usually use asynchronous replication across data centers.

For replication within the data center, we can get strong consistency with good performance. We can compromise strong consistency across data center replication to achieve better availability (see CAP and PACELC theorems)

Consistency

It's possible to write data to cache servers in a synchronous or asynchronous mode. In the case of caching, the asynchronous mode is favoured for improved performance. Consequently, our caching system suffers from inconsistencies. Alternatively, strong consistency comes from synchronous writing, but this increases the overall latency, and the performance takes a hit.

Inconsistency can also arise from faulty configuration files and services. Imagine a scenario where a cache server is down during a write operation, and a read operation is performed on it just after its recovery. We can avoid such scenarios for any joining or rejoining server by not allowing it to serve requests until it's reasonably sure that it's up to date.

Calculations

To get an idea of how important the eviction algorithm is, let's assume the following:

- Cache hit service time (99.9*th* percentile): 5 ms
- Cache miss service time (99.9*th* percentile): 30 ms (this includes time to get the data from the database and set the cache)

Let's assume we have a 10% cache miss rate using the most frequently used (MFU) algorithm, whereas we have a 5% cache miss rate using the LRU algorithm. Then, we use the following formula:

EAT = *Ratiohit* x *Timehit* + *Ratiomiss* x *Timemiss*

Here, this means the following:

EAT: Effective access time.

Ratiohit: The percentage of times a cache hit will occur.

Ratiomiss: The percentage of times a cache miss will occur.

Timehit: Time required to serve a cache hit.

Timemiss: Time required to serve a cache miss.

For MFU, we see the following:

```
EAT = 0.90 x 5 milliseconds + 0.10 x 30 milliseconds = 0.0045 + 0.003 = 0.0075
```

For LRU, we see the following:

```
EAT = 0.95 x 5 milliseconds + 0.05 x 30 milliseconds =  0.00475 + 0.0015 = 0.00
```

The numbers above highlight the importance of the eviction algorithm to increase the cache hit rate. Each application should conduct an empirical study to determine the eviction algorithm that gives better results for a specific workload.

Distributed Systems Distributed Cache System Design Interview
Software Architecture Software Architect

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium